ApplicationListener：监听容器中发布的事件。事件驱动模型开发；

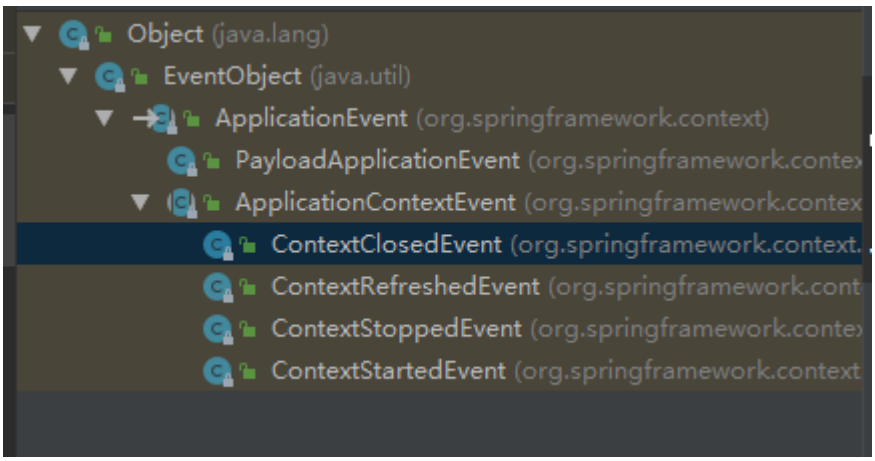
ApplicationListener监听器的接口:

```
public interface ApplicationListener<E extends ApplicationEvent> extends EventListener {

    void onApplicationEvent(E event);

}
```

注意:ApplicationListener监听器的实现类只能监听 ApplicationEvent 及其下面的子事件



注:ApplicationEvent 的子类,ContextRefreshedEvent(容器刷新完成事件)这个用的多,ContextClosedEvent(容器关闭事件)这个用的也多

只要事件发布 监听器就可以监听到

自定义事件发布的步骤:

1）、写一个监听器（ApplicationListener实现类）来监听某个事件（ApplicationEvent及其子类）

2）、把监听器加入到容器；

3）、只要容器中有相关事件的发布，我们就能监听到这个事件；

例如:ContextRefreshedEvent：容器刷新完成（所有bean都完全创建）会发布这个事件；

ContextClosedEvent：关闭容器会发布这个事件；

4）、发布一个事件：

拿到一个容器直接调用其的publishEvent()方法

applicationContext.publishEvent();

整套事件发布与监听的过程:

以ContextRefreshedEvent这个事件为例进行说明

还记得我们在容器刷新哪里详细说了一个initApplicationEventMulticaster()方法吗?

1.给容器中注册一个多播器

```
//给容器中注册了一个派发器(多播器)
protected void initApplicationEventMulticaster() {
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    if (beanFactory.containsLocalBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME)) {
        this.applicationEventMulticaster =
                beanFactory.getBean(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
ApplicationEventMulticaster.class);
        if (logger.isDebugEnabled()) {
            logger.debug("Using ApplicationEventMulticaster [" + this.applicationEventMulticaster
+ "]");
        }
    }
    else {
        this.applicationEventMulticaster = new SimpleApplicationEventMulticaster(beanFactory);
        beanFactory.registerSingleton(APPLICATION_EVENT_MULTICASTER_BEAN_NAME,
this.applicationEventMulticaster);
        if (logger.isDebugEnabled()) {
            logger.debug("Unable to locate ApplicationEventMulticaster with name '" +
                    APPLICATION_EVENT_MULTICASTER_BEAN_NAME +
                    "': using default [" + this.applicationEventMulticaster + "]");
        }
    }
}
```

还记得我们在容器刷新哪里详细说了一个registerListeners()方法吗?

2.注册监听器

```
//将监听器注册到多播器中
protected void registerListeners() {
    // Register statically specified listeners first.
    //将监听器注册到多播器中
    for (ApplicationListener<?> listener : getApplicationListeners()) {
        getApplicationEventMulticaster().addApplicationListener(listener);
    }

    // Do not initialize FactoryBeans here: We need to leave all regular beans
    // uninitialized to let post-processors apply to them!
    //通过名称拿到所有的监听器 将其注册到多播器中
    String[] listenerBeanNames = getBeanNamesForType(ApplicationListener.class, true, false);
    for (String listenerBeanName : listenerBeanNames) {
        getApplicationEventMulticaster().addApplicationListenerBean(listenerBeanName);
    }

    // Publish early application events now that we finally have a multicaster...
    Set<ApplicationEvent> earlyEventsToProcess = this.earlyApplicationEvents;
    this.earlyApplicationEvents = null;
    if (earlyEventsToProcess != null) {
        for (ApplicationEvent earlyEvent : earlyEventsToProcess) {
            getApplicationEventMulticaster().multicastEvent(earlyEvent);
        }
```

```
    }
}
```

3.事件发布并调用多播器派发事件

事件发布在整个容器刷新的阶段的最后一步finishRefresh()中

1）、获取事件的多播器（派发器）：getApplicationEventMulticaster() 2）、multicastEvent派发事件： 3）、获取到所有的ApplicationListener； for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) { 1）、如果有Executor，可以支持使用Executor进行异步派发； Executor executor = getTaskExecutor(); 2）、否则，同步的方式直接执行listener方法；invokeListener(listener, event); 拿到listener回调onApplicationEvent方法；

```
protected void finishRefresh() {
    // Initialize lifecycle processor for this context.
    initLifecycleProcessor();

    // Propagate refresh to lifecycle processor first.
    getLifecycleProcessor().onRefresh();

    // Publish the final event.
    //发布一个ContextRefreshedEvent事件
    publishEvent(new ContextRefreshedEvent(this));

    // Participate in LiveBeansView MBean, if active.
    LiveBeansView.registerApplicationContext(this);
}
//发布事件
protected void publishEvent(Object event, ResolvableType eventType) {
    Assert.notNull(event, "Event must not be null");
    if (logger.isTraceEnabled()) {
        logger.trace("Publishing event in " + getDisplayName() + ": " + event);
    }

    // Decorate event as an ApplicationEvent if necessary
    ApplicationEvent applicationEvent;
    if (event instanceof ApplicationEvent) {
        applicationEvent = (ApplicationEvent) event;
    }
    else {
        applicationEvent = new PayloadApplicationEvent<Object>(this, event);
        if (eventType == null) {
            eventType = ((PayloadApplicationEvent)applicationEvent).getResolvableType();
        }
    }

    // Multicast right now if possible - or lazily once the multicaster is initialized
    if (this.earlyApplicationEvents != null) {
        this.earlyApplicationEvents.add(applicationEvent);
    }
    else {
        //拿到事件派发器(多播器)并派发事件
        getApplicationEventMulticaster().multicastEvent(applicationEvent, eventType);
    }

    // Publish event via parent context as well...
    if (this.parent != null) {
```

```
        if (this.parent instanceof AbstractApplicationContext) {
            ((AbstractApplicationContext) this.parent).publishEvent(event, eventType);
        }
        else {
            this.parent.publishEvent(event);
        }
    }
}
//派发事件
public void multicastEvent(final ApplicationEvent event, ResolvableType eventType) {
    ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event));
    //拿到所有的事件监听器 循环遍历
    for (final ApplicationListener<?> listener : getApplicationListeners(event, type)) {
        //如果可以异步派发事件 就使用多线程异步派发事件
        Executor executor = getTaskExecutor();
        if (executor != null) {
            executor.execute(new Runnable() {
                @Override
                public void run() {
                    invokeListener(listener, event);
                }
            });
        }
        else {
            //如果没有直接执行监听的器的触发方法
            invokeListener(listener, event);
        }
    }
}
```

有没有感觉很熟悉？像不像一个设计模式？没错，就是观察者 spring的事件机制也是用的观察者模式

说一下@EventListener这个注解

示例代码:

```
@Service
public class UserService {

    @EventListener(classes={ApplicationEvent.class})
    public void listen(ApplicationEvent event){
        System.out.println("UserService...监听到的事件："+event);
    }

}
```

原理：使用EventListenerMethodProcessor处理器来解析方法上的@EventListener；

```
public class EventListenerMethodProcessor implements SmartInitializingSingleton,
ApplicationContextAware {

    protected final Log logger = LogFactory.getLog(getClass());

    private ConfigurableApplicationContext applicationContext;

    private final EventExpressionEvaluator evaluator = new EventExpressionEvaluator();
```

```
    private final Set<Class<?>> nonAnnotatedClasses =
            Collections.newSetFromMap(new ConcurrentHashMap<Class<?>, Boolean>(64));


    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException
{
        Assert.isTrue(applicationContext instanceof ConfigurableApplicationContext,
                "ApplicationContext does not implement ConfigurableApplicationContext");
        this.applicationContext = (ConfigurableApplicationContext) applicationContext;
    }
    //SmartInitializingSingleton接口的afterSingletonsInstantiated方法
    @Override
    public void afterSingletonsInstantiated() {
        List<EventListenerFactory> factories = getEventListenerFactories();
        String[] beanNames = this.applicationContext.getBeanNamesForType(Object.class);
        for (String beanName : beanNames) {
            if (!ScopedProxyUtils.isScopedTarget(beanName)) {
                Class<?> type = null;
                try {
                    type =
AutoProxyUtils.determineTargetClass(this.applicationContext.getBeanFactory(), beanName);
                }
                catch (Throwable ex) {
                    // An unresolvable bean type, probably from a lazy bean - let's ignore it.
                    if (logger.isDebugEnabled()) {
                        logger.debug("Could not resolve target class for bean with name '" +
beanName + "'", ex);
                    }
                }
                if (type != null) {
                    if (ScopedObject.class.isAssignableFrom(type)) {
                        try {
                            type =
AutoProxyUtils.determineTargetClass(this.applicationContext.getBeanFactory(),
                                    ScopedProxyUtils.getTargetBeanName(beanName));
                        }
                        catch (Throwable ex) {
                            // An invalid scoped proxy arrangement - let's ignore it.
                            if (logger.isDebugEnabled()) {
                                logger.debug("Could not resolve target bean for scoped proxy '" +
beanName + "'", ex);
                            }
                        }
                    }
                    try {
                        processBean(factories, beanName, type);
                    }
                    catch (Throwable ex) {
                        throw new BeanInitializationException("Failed to process @EventListener "
+
                                "annotation on bean with name '" + beanName + "'", ex);
                    }
                }
            }
        }
```

```
    }


    /**
     * Return the {@link EventListenerFactory} instances to use to handle
     * {@link EventListener} annotated methods.
     */
    protected List<EventListenerFactory> getEventListenerFactories() {
        Map<String, EventListenerFactory> beans =
this.applicationContext.getBeansOfType(EventListenerFactory.class);
        List<EventListenerFactory> allFactories = new ArrayList<EventListenerFactory>
(beans.values());
        AnnotationAwareOrderComparator.sort(allFactories);
        return allFactories;
    }

    protected void processBean(final List<EventListenerFactory> factories, final String beanName,
final Class<?> targetType) {
        if (!this.nonAnnotatedClasses.contains(targetType)) {
            Map<Method, EventListener> annotatedMethods = null;
            try {
                annotatedMethods = MethodIntrospector.selectMethods(targetType,
                        new MethodIntrospector.MetadataLookup<EventListener>() {
                            @Override
                            public EventListener inspect(Method method) {
                                return AnnotatedElementUtils.findMergedAnnotation(method,
EventListener.class);
                            }
                        });
            }
            catch (Throwable ex) {
                // An unresolvable type in a method signature, probably from a lazy bean - let's
ignore it.
                if (logger.isDebugEnabled()) {
                    logger.debug("Could not resolve methods for bean with name '" + beanName +
"'", ex);
                }
            }
            if (CollectionUtils.isEmpty(annotatedMethods)) {
                this.nonAnnotatedClasses.add(targetType);
                if (logger.isTraceEnabled()) {
                    logger.trace("No @EventListener annotations found on bean class: " +
targetType.getName());
                }
            }
            else {
                // Non-empty set of methods
                for (Method method : annotatedMethods.keySet()) {
                    for (EventListenerFactory factory : factories) {
                        if (factory.supportsMethod(method)) {
                            Method methodToUse = AopUtils.selectInvocableMethod(
                                    method, this.applicationContext.getType(beanName));
                            ApplicationListener<?> applicationListener =
                                    factory.createApplicationListener(beanName, targetType,
methodToUse);
                            if (applicationListener instanceof ApplicationListenerMethodAdapter) {
                                ((ApplicationListenerMethodAdapter) applicationListener)
```

```
                                        .init(this.applicationContext, this.evaluator);
                            }
                            this.applicationContext.addApplicationListener(applicationListener);
                            break;
                        }
                    }
                }
                if (logger.isDebugEnabled()) {
                    logger.debug(annotatedMethods.size() + " @EventListener methods processed on
bean '" +
                            beanName + "': " + annotatedMethods);
                }
            }
        }
    }

}
```

EventListenerMethodProcessor这个又是在哪儿执行的呢?

EventListenerMethodProcessor实现了SmartInitializingSingleton这个接口 这个接口是在所有单实例创建完毕后进行触发执行

在容器刷新时的finishBeanFactoryInitialization() -> beanFactory.preInstantiateSingletons()

```
public void preInstantiateSingletons() throws BeansException {
    if (this.logger.isDebugEnabled()) {
        this.logger.debug("Pre-instantiating singletons in " + this);
    }

    // Iterate over a copy to allow for init methods which in turn register new bean definitions.
    // While this may not be part of the regular factory bootstrap, it does otherwise work fine.
    List<String> beanNames = new ArrayList<String>(this.beanDefinitionNames);

    // Trigger initialization of all non-lazy singleton beans...
    //即将要创建的所有的bean
    for (String beanName : beanNames) {
        RootBeanDefinition bd = getMergedLocalBeanDefinition(beanName);
        if (!bd.isAbstract() && bd.isSingleton() && !bd.isLazyInit()) {
            if (isFactoryBean(beanName)) {
                final FactoryBean<?> factory = (FactoryBean<?>) getBean(FACTORY_BEAN_PREFIX +
beanName);
                boolean isEagerInit;
                if (System.getSecurityManager() != null && factory instanceof SmartFactoryBean) {
                    isEagerInit = AccessController.doPrivileged(new PrivilegedAction<Boolean>() {
                        @Override
                        public Boolean run() {
                            return ((SmartFactoryBean<?>) factory).isEagerInit();
                        }
                    }, getAccessControlContext());
                }
                else {
                    isEagerInit = (factory instanceof SmartFactoryBean &&
                            ((SmartFactoryBean<?>) factory).isEagerInit());
                }
                if (isEagerInit) {
```

```
                    getBean(beanName);
                }
            }
            else {
                getBean(beanName);
            }
        }
    }
    //此时bean已经创建完毕
    // Trigger post-initialization callback for all applicable beans...
    //再次拿到所有的单实例bean
    for (String beanName : beanNames) {
        Object singletonInstance = getSingleton(beanName);
        //如果bean为SmartInitializingSingleton
        if (singletonInstance instanceof SmartInitializingSingleton) {
            final SmartInitializingSingleton smartSingleton = (SmartInitializingSingleton)
singletonInstance;
            //如果是可以异步执行
            if (System.getSecurityManager() != null) {
                AccessController.doPrivileged(new PrivilegedAction<Object>() {
                    @Override
                    public Object run() {
                        //创建线程异步执行SmartInitializingSingleton接口的afterSingletonsInstantiated
方法
                        smartSingleton.afterSingletonsInstantiated();
                        return null;
                    }
                }, getAccessControlContext());
            }
            else {
                //同步执行SmartInitializingSingleton接口的afterSingletonsInstantiated方法
                smartSingleton.afterSingletonsInstantiated();
            }
        }
    }
}
```